Adding Traits to (Statically Typed) Languages

Oscar Nierstrasz, Stéphane Ducasse, Stefan Reichhart and Nathanael Schärli

Institut für Informatik und Angewandte Mathematik
University of Bern, Switzerland

IAM-05-006

December 1, 2005

Abstract

Traits offer a fine-grained mechanism for composing classes in object-oriented languages from reusable components, while avoiding the fragility problems introduced by multiple inheritance and mixins. Although traits were developed in the context of dynamically typed languages, they would also offer clear benefits for statically typed languages like Java and C#. This report summarizes the issues raised when integrating traits into such languages. We examine traits in the context of the statically typed languages FEATHERWEIGHT JAVA, C# and C++.

CR Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features—*Inheritance*

1 Introduction

Traits were introduced [16, 17] as a simple programming language mechanism for incrementally composing classes from small, reusable components, while avoiding problems of fragility in the class hierarchy that arise with approaches based on mixins or multiple inheritance. Traits are essentially sets of methods, divorced from any instance variables or a superclass. Composite traits may be composed from subtraits using the trait composition operators.

Initial experiences using traits in SMALLTALK to refactor complex class hierarchies have been very promising [1], and the question naturally arises, how can we apply traits to other languages. This question is especially interesting if we consider statically-typed languages like Java and C# because adding traits to such a language requires integrating them into its static type system.

Besides the question of what is the relationship between traits and types, such an integration also raises the question of how to type trait methods. Unlike their dynamically typed counterparts, statically typed languages require the programmer to define a static type for the arguments and the return value of each trait method. But how is it possible to do this in a way that keeps the trait generic enough to be applied to many different classes, some of which may not be known at compile time? Does it require an extension of the underlying type system?

While the questions related to static typing are very interesting, there are also many other issues and trade-offs that arise when traits are added to another language. For example, it is usually easier to implement traits by compiling them away, but this also means that they are not represented in the code that actually gets executed, which makes it harder to use features such as debuggers and runtime reflection. Also, depending on the implementation strategy, there may or may not be a duplication of the executable code corresponding to traits.

The goal of this paper is to provide the reader with a road map of issues and possible strategies related to the integration of traits into (statically typed) languages. While some of these strategies are based on formal models and are quite general, others are more pragmatic and language-specific. We also present the strategies taken by existing implementation of traits and adaptations of traits, and we analyze how they address the important issues.

The rest of this paper is structured as follows: In Section 2, we give a brief introduction of traits and present an example. In Section 3, we give an overview of different issues that arise when traits are added to programming languages. In Section 4, we present a formal model for a flattening-based strategy of adding traits to a statically typed programming language. While this model is very simple and generic, it omits many of the more sophisticated issues related to the integration of traits into a static type system. In Section 5, we therefore sketch how two extensions of this model can lead to reasonable design choices for these issues. In Section 6, we examine how this formal model can be applied to the language C#, and present our implementation. In Section 7, we investigate how traits can be simulated in C++ using templates and (virtual) multiple inheritance, and we discuss the consequences of such a strategy. In Section 8, we present our original implementation of traits in the Smalltalk dialect Squeak and evaluate it against the identified issues. In Section 9, we present and analyze the strategies taken by existing implementation of traits and adaptations of traits.

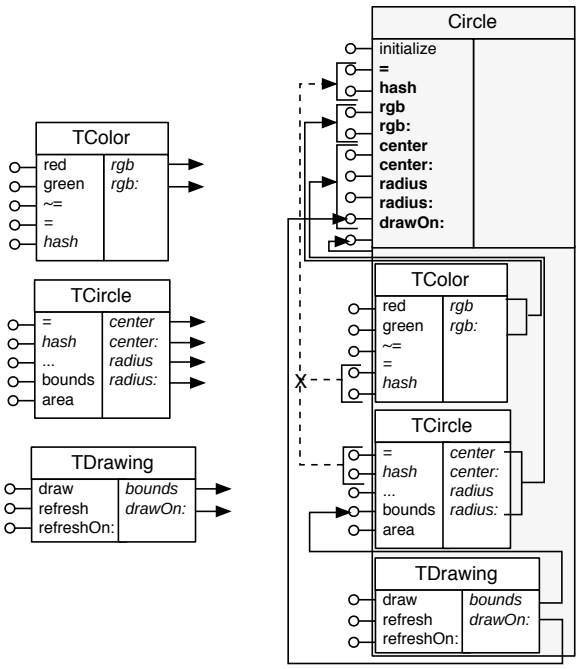

Figure 1: Class Circle is composed from traits TCircle, TColor and TDrawing.

2 Traits in a Nutshell

Traits [17] are essentially groups of methods that serve as building blocks for classes and are primitive units of code reuse. As such, they allow one to factor out common behavior and form an intermediate level of abstraction between single methods and complete classes. A trait consists of *provided methods* that implement its behavior, and of *required methods* that parameterize the provided behavior. Traits do not specify any instance variables, and the methods provided by traits never directly access instance variables. Instead, required methods can be mapped to state when the trait is used by a class.

With traits, the behavior of a class is specified as the composition of traits and some *glue methods* that are implemented at the level of the class. These glue methods connect the traits together and can serve as accessor for the necessary state. The semantics of such a class is defined by the following three rules:

- *Class methods take precedence over trait methods.* This allows the glue methods defined in the class to override equally named methods provided by the traits.
- *Flattening property.* A trait method which is not overridden by a client class has the same semantics as the same method implemented directly in that class.
- *Composition order is irrelevant.* All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

Because the composition order is irrelevant, a *conflict* arises if we combine two or more traits that provide identically named methods that do not originate from the same trait. Traits enforce explicit resolution of conflicts by implementing a glue method at the level of the class that overrides the conflicting methods, or by *method exclusion*, which allows one to exclude the conflicting method from all but one trait. In addition traits support *method aliasing*. The programmer can introduce an additional name for a method provided by a trait to obtain access to a method that would otherwise be unreachable, for example, because it has been overridden.

Example: Geometric Objects. Suppose we want to represent a graphical object such as a circle or square that is drawn on a canvas. Such a graphical object can be decomposed into three reusable aspects — its geometry, its color and the way that it is drawn on a canvas.

Figure 1 shows this for the case of a Circle class composed from traits TCircle, TColor and TDrawing:

- TCircle defines the geometry of a circle: it requires the methods center, center:, radius, and radius: and provides methods such as bounds, hash, and =.
- TDrawing requires the methods drawOn: bounds and provides the methods draw, refresh, and refreshOn:.
- TColor requires the methods rgb, rgb: and provides all kind of methods manipulating colors. We only show the methods hash and = as they will conflict with others at composition time.

The class Circle specifies three instance variables center, radius, and rgb and their respective accessor methods. It is composed from the three traits TDrawing, TCircle, and TColor. As there is a conflict for the methods hash and = between the traits TCircle and TColor, we must alias those methods in both traits to be able to access them in the methods hash and = of the class Circle resolving the conflicts.

3 Issues

As we have pointed out in the previous section, traits obey the flattening property. This means that a program written with traits can be translated into a semantically equivalent program without traits by inlining all the traits in the classes where they are used.

As a consequence, there is a very simple and generic strategy for adding traits to a language L, which consists of the following two steps.

1. Create a language L_T that differs from L only in that it has an extended syntax that allows on to define and use traits.

2. Write a translator that takes a program written in L_T and then inlines all the traits to yield a semantically equivalent trait-less program in the base language L.

This strategy has several advantages: it is very simple and generic, it preserves the semantics (because of the flattening property), and it does not require any changes to the compilers (and potential virtual machines) for the language L. However, it also means that traits are treated as syntactic sugar and completely disappear during the translation, which leads to several problems.

Besides the fact that the translation duplicates the code of each trait in all the classes where it is used, it also leads to inconsistencies between the source code (written in L_T) and the code that actually gets executed at runtime. As a consequence, one can for example not use runtime reflection to reason about traits, and if reflection is used to reason about classes, the absence of traits may lead to a result that is different than what one would expect. A similar effect also occurs when debugging a program.

Even more issues arise when this strategy is applied to a statically typed language. This is because in this case, one also has to think about how to integrate traits into the static type system to get an adequate expressiveness in the new langauge L_T.

In the rest of this section, we give an overview of important issues that arise when traits are added to a language and outline design decisions that could be used to address them.

3.1 Representing Traits in the Language

For a smooth and complete integration, traits should be represented in the language. Ideally, representations of traits should exist at both compile-time and runtime. This has several reasons.

- *Compilation.* Having a compile-time entity for traits is the basis for being able to compile traits separately, *i.e.*, independently of the classes where they are used. This not only a allows on to detect errors in traits at an early stage (and independent of potential errors in the classes where a trait is used), but it is also the basis for sharing the trait code among multiple classes.

If traits cannot be compiled separately from the classes where they are used, it should at least be possible for a programmer to easily see whether a certain compile-time error is caused in a class or a trait that is used by the class being compiled.

- *Debugging.* Traits should be represented in the debugger, so that a programmer can easily map code being executed to the actual source code (written with traits).

- *Runtime reflection.* Many programming language support a form of runtime reflection that allows one to reflect and sometimes even manipulate the program being executed. If traits are added to a language, it is important that they are correctly represented in the reflective infrastructure of the language, so that one can for example ask which methods are provided by a certain trait or which traits are used by a certain class.

While the advantages of representing traits in the language are quite general, the question of how to achieve such representations strongly depends on the type of programming language, and there are a wide variety of design decisions.

On one hand, traits are similar to (abstract) classes, and so it looks like a good idea to take advantage of this similarity and represent traits in a similar way. In languages featuring multiple inheritance and templates or other macro facilities (such as C++), it may even be possible to represent traits as special classes, that can then be arranged in inheritance hierarchies that exhibit the composition semantics known from traits.

In statically typed languages, another important issue is the relationship between traits and types. In most of these languages, every class also defines a type, and so one could argue that also every trait should define a type. Because a class can use multiple traits, this would add a form of multiple subtyping to the language. However, many of the recent languages also support interfaces (as promoted by Java), which define types as well and are specifically used for multiple subtyping. Since traits also bear a similarity to interfaces, this poses the question whether it is actually necessary to have all these different but also similar forms of types.

3.2 Typing Trait Methods

Another issue that arises when traits are implemented in a statically typed language is that it may not always be clear how their methods should be typed so that they can be reused across multiple classes.

As an illustration, consider the trait TLinkable that bundles together the behavior of an element in a linked list and is used in classes such as Link and Process [1]. This trait provides, amongst others, the methods includes:, checkAndSetNext: and reverse. While includes: checks whether the argument link is included in the linked list starting at the receiver, the method checkAndSetNext: sets the next field of the receiver to the link that is passed as an argument, but only if this does not cause a cycle. Finally, the method reverse reverses the linked list. Figure 2 shows the implementation of this trait in Smalltalk.

Because Smalltalk is dynamically typed, the trait TLinkable can be immediately used in the class Link, the class Process, and any other class that is linkable. The only condition is that these classes provide the two required methods getNext and setNext:, which get and set the next element of the list.

[1] Smalltalk processes are links so that the scheduler can keep them in linked lists.

```
Trait named: #TLinkable

includes: other
    | tail |
    self == other ifTrue: [↑ true].
    tail := self getNext.
    tail ifNil: [↑ false].
    ↑ tail includes: other

checkAndSetNext: other
    (other includes: self) ifTrue: [↑ false].
    self setNext: other.
    ↑ true.

reverse
    | result list temp |
    result := nil.
    list := self.
    [list notNil] whileTrue: [
        temp := list getNext.
        list setNext: result.
        result := list.
        list := temp.
    ].
    ↑ result.

getNext
    self requirement

setNext: other
    self requirement
```

Figure 2: The trait TLinkable in the dynamically typed language Smalltalk

Now assume that we would like to write the same trait in a statically typed language such as Java and C#. This immediately raises the question of what static type should be used for the arguments, the return values, and the temporary variables of the methods defined in the trait TLinkable so that this trait can be used for both Link and Process as well as any other class that is linkable.

Types of Arguments. Regarding the argument type of includes:, a reasonable answer would be that the chosen type should allow every linkable object to be passed as an argument. In a language where every trait also defines its own type and trait composition establishes a subtype relationship, one might for example use the type TLinkable as the argument type. Alternatively, in a language where traits do not define types, one could use as an argument type a separate interface ILinkable, that would then need to be implemented by all linkable classes.

Types of Return Values. When it comes to the other types, things are more problematic. As an example, consider the return type of the methods reverse and getNext. Assuming that TLinkable is used only for homogeneous lists, the methods reverse and getNext should return an instance of whatever class they are called on. In particular, this means that an instance of Link (Process) should be returned when these methods are called on a Link (Process).

What makes this situation difficult is that the return types of these methods are in fact parametric; *i.e.*, they depend on the class to which the trait TLinkable is finally applied. Therefore, using an interface such as ILinkable as the return type does not solve our problem because it would only allow a common subset of all the methods in Link and Process to be called on the return values.

The same problem also applies to the argument of the methods setNext: and checkAndSetNext: as well as to the temporary variables used in the method reverse. If we for example declared the type of these temporary variables to be ILinkable, the type of the list elements would be changed when the list is reversed.

The problem could be addressed using parametric polymorphism as provided by the generics mechanism available in C# and Java 1.5 (or later), because it allows us to write the trait TLinkable with a type parameter that is then used for the return values, the arguments, and the temporary variables of these methods.

Another approach would be to address this problem by reifying the class that actually uses the trait. This means that the language would get extended with a construct to refer to the class where a trait will eventually be used. Using this construct, one could write the trait TLinkable without the explicit use of generics, which leads to a simpler and more concise solution.

Overloading. Having typed methods in traits also means that, depending on the semantics of the underlying language, we might need to deal with method overloading. For example, trait composition can generate cases of ambiguous overloading (*i.e.*, when the static type system cannot uniquely determine which method to dispatch), which needs to be distinguished from method conflicts. Another complication is that in presence of overloading, plain method names are not enough to uniquely identify a method. In case of aliases, this for example poses the question whether the new method signature must be identical to the old one.

3.3 Adapting the Compilation and Execution Mechanisms

The most fundamental issue for adding traits to a language is the question of how to adapt the compilation and execution mechanisms of the language so that code written with traits is correctly executed. Ideally, these mechanisms should be adapted so that they satisfy the following two properties.

- *Small Programs.* Traits should not only allow one to reuse source code, but also to reuse executable code. This means that a program should contain the executable code for each trait only once; all classes (and traits) using a certain trait should refer to the exact same code.
- *High Execution Performance.* When traits are added to a language, it should have no (negative) effects on execution speed. This means that a program written with trait should be executed (at least) as fast as the corresponding flattened program that does not contain any traits.

In reality it is not only hard to achieve both of these properties together, but it may also require a significant engineering effort. As an example, assume that we want to add traits to a language that gets executed on a virtual machine. Using the simple flattening based strategy outlined at the beginning of this section, one only has to extend the compiler so that it first inlines all the traits. This has the advantage that no changes to the virtual machine are necessary, and that the execution performance is essentially the same as before. However, it also means that the executable code of each trait is duplicated in all classes where it is used.

Alternatively, one could modify the compiler *and* the virtual machine so that traits are compiled separately and the method lookup algorithm does not only take the inheritance hierarchy but also traits hierarchies into account. However, this requires more engineering work and is likely to result in slower execution speed. Furthermore, it means that code with traits cannot be executed on earlier virtual machines.

4 A Flattening-based Calculus for Traits

In the previous section, we have outlined how flattening can be used as a simple and generic strategy for adding traits to a programming language. In this section, we formalize this strategy. For simplicity, we do not use a real programming language as the basis; instead we use FJ, which is a minimal core calculus for the Java programming language [8]. Because FJ omits many of the more specific issues such as concurrency and reflection, this model is quite general and also applies to the core of similar languages such as C#.

Given the base language FJ, the definition of the extended language FTJ consists of two steps. First, we define the syntax of FTJ, which is an extension of the FJ syntax with the necessary constructs for defining and using traits. Second, we define the semantics of FTJ by specifying a flattening function that translates any FTJ program to an FJ program.

4.1 Featherweight Trait Java (FTJ)

Figure 3 shows the syntax of FTJ, which is borrowed from Liquori and Spiwack, who defined a calculus that is a conservative extension of FJ with minimal syntactic and semantic changes

CL	::=	class C ◁ C uses $\overline{\texttt{TA}}$ {$\overline{\texttt{C}}$ $\overline{\texttt{f}}$;K $\overline{\texttt{M}}$}	*Classes*
TL	::=	trait T uses $\overline{\texttt{TA}}$ {$\overline{\texttt{M}}$}	*Traits*
TA	::=	T | TA with {m@n} | TA minus {m}	*Trait expressions*
K	::=	C($\overline{\texttt{C}}$ $\overline{\texttt{f}}$) {super($\overline{\texttt{f}}$);this.$\overline{\texttt{f}}$=$\overline{\texttt{f}}$;}	*Constructors*
M	::=	C m($\overline{\texttt{C}}$ $\overline{\texttt{x}}$) {↑e;}	*Methods*
e	::=	x | e.f | e.m($\overline{\texttt{e}}$) | new C($\overline{\texttt{e}}$) | (C)e	*Expressions*

Figure 3: FTJ Syntax.

to accommodate traits [10]. The only differences with the syntax of FJ are the modification of class definitions to include a sequence of *used traits* $\overline{\texttt{TA}}$, and the addition of syntax for trait definitions (TL) and trait expressions (TA). As in FJ, the notation $\overline{\texttt{C}}$ denotes a possible empty sequence of elements C (with or without commas, as appropriate; • represents the empty sequence.) For the sake of conciseness we abbreviate the keywords `extends` to the symbol ◁ and the keyword `return` to the symbol ↑.

Traits cannot specify any instance variables ($\overline{\texttt{f}}$), and the methods provided by traits never directly access instance variables. Instead, required methods are mapped to state when the trait is used by a class.

With traits, the behavior of a class is specified as the composition of traits and some *glue methods* ($\overline{\texttt{M}}$) that are implemented at the level of the class (CL) or the composite trait (TL). These glue methods connect the traits together and can serve as accessor for the necessary state.

The operational semantics of FTJ specifies a modified method lookup algorithm that ensures that methods of a class C take precedence over methods provided by any of the used traits $\overline{\texttt{TA}}$. Similarly, methods of a named trait T take precedence over methods provided by subtraits $\overline{\texttt{TA}}$ used by T.

Because the composition order is irrelevant, a *conflict* arises if we combine two or more traits that provide identically named methods that do not originate from the same trait. $\overline{\texttt{TA}}$ is a composition of traits $\texttt{T}_i$, possibly giving rise to conflicts. Conflicts may be resolved by overriding them with glue methods $\overline{\texttt{M}}$ in the class using $\overline{\texttt{TA}}$, or by excluding the conflicting methods. `TA minus {m}` removes the method named m from the trait expression TA.

In addition traits allow *method aliasing*. The programmer can introduce an additional name for a method provided by a trait to obtain access to a method that would otherwise be unreachable because it has been overridden. `TA with {m@n}` defines m to be an alias for the existing method named n. (Note that the aliasing syntax of FTJ (`m@n`) puts the new name n *after* the existing method name m, whereas the aliasing operator (→) expects its arguments in the reverse order.)

4.2 Flattening FTJ

We have previously developed a simple set-theoretic model of traits [15]. The goals of this model were to define the trait composition operators, to give an operational account of method lookup (particularly `self`- and `super`-sends), and to develop a notion of equivalence for traits. The model further makes precise the notion of method *conflicts* arising during trait composition, and the notion that a class constructed using traits can always be flattened into

$$lookup(\mathtt{m}, \overline{\mathtt{M}}) \stackrel{\text{def}}{=} \begin{cases} M & \text{if } M = \mathtt{C\ m(}\overline{\mathtt{C}}\ \mathtt{x)\ \{}\uparrow\mathtt{e;\}} \in \overline{\mathtt{M}} \\ \bot & \text{otherwise} \end{cases} \quad (1)$$

$$extract(X, \overline{\mathtt{M}}) \stackrel{\text{def}}{=} \bigwedge_{\mathtt{m} \in X} lookup(\mathtt{m}, \overline{\mathtt{M}}) \quad (2)$$

$$mNames(\overline{\mathtt{M}}) \stackrel{\text{def}}{=} \{\mathtt{m} \mid lookup(\mathtt{m}, \overline{\mathtt{M}}) \neq \bot\} \quad (3)$$

$$\frac{\mathtt{trait\ T\ uses}\ \overline{\mathtt{TA}}\ \mathtt{\{}\overline{\mathtt{M}}\mathtt{\}}}{local(\mathtt{T}) = \overline{\mathtt{M}}} \qquad \frac{\mathtt{trait\ T\ uses}\ \overline{\mathtt{TA}}\ \mathtt{\{}\overline{\mathtt{M}}\mathtt{\}}}{subtraits(\mathtt{T}) = \overline{\mathtt{TA}}} \quad (4)$$

$$\overline{\mathtt{M}} - \mathtt{m} \stackrel{\text{def}}{=} \overline{\mathtt{M}} \backslash lookup(\mathtt{m}, \overline{\mathtt{M}}) \quad (5)$$

$$\overline{\mathtt{M}}_1 \triangleright \overline{\mathtt{M}}_2 \stackrel{\text{def}}{=} \overline{\mathtt{M}}_1, (\overline{\mathtt{M}}_2 \backslash extract(mNames(\overline{\mathtt{M}}_1), \overline{\mathtt{M}}_2)) \quad (6)$$

$$\overline{\mathtt{M}}[\mathtt{n} \rightarrow \mathtt{m}] \stackrel{\text{def}}{=} \begin{cases} (\overline{\mathtt{M}} \backslash lookup(\mathtt{n}, \overline{\mathtt{M}})), conflict(\mathtt{n}) & \text{if } lookup(\mathtt{n}, \overline{\mathtt{M}}) \neq \bot \\ \overline{\mathtt{M}}, \mathtt{C\ n(}\overline{\mathtt{C}}\ \overline{\mathtt{x}}\mathtt{)\{}\uparrow\mathtt{e;\}} & \text{else if } \mathtt{C\ m(}\overline{\mathtt{C}}\ \overline{\mathtt{x}}\mathtt{)\{}\uparrow\mathtt{e;\}} \in \overline{\mathtt{M}} \\ \overline{\mathtt{M}} & \text{otherwise} \end{cases} \quad (7)$$

$$mBodies(\overline{\mathtt{M}}_1, \overline{\mathtt{M}}_2) \stackrel{\text{def}}{=} extract(mNames(\overline{\mathtt{M}}_1) \backslash mNames(\overline{\mathtt{M}}_2), \overline{\mathtt{M}}_1) \quad (8)$$

$$broken(\overline{\mathtt{M}}_1, \overline{\mathtt{M}}_2) \stackrel{\text{def}}{=} (mNames(\overline{\mathtt{M}}_1) \cap mNames(\overline{\mathtt{M}}_2)) \backslash mNames(\overline{\mathtt{M}}_1 \cap \overline{\mathtt{M}}_2) \quad (9)$$

$$\overline{\mathtt{M}}_1 + \overline{\mathtt{M}}_2 \stackrel{\text{def}}{=} mBodies(\overline{\mathtt{M}}_1, \overline{\mathtt{M}}_2), mBodies(\overline{\mathtt{M}}_2, \overline{\mathtt{M}}_1), (\overline{\mathtt{M}}_1 \cap \overline{\mathtt{M}}_2), \bigwedge\{conflict(\mathtt{m}) \mid \mathtt{m} \in broken(\overline{\mathtt{M}}_1, \overline{\mathtt{M}}_2)\} \quad (10)$$

$$\text{where } conflict(\mathtt{m}) = \mathtt{Object\ m()\ \{}\uparrow\bot\mathtt{;\ \}}$$

Figure 4: Composition operators for FTJ

one that does not use traits.

The trait model defines *method dictionaries* as mappings from method signatures to method bodies. A *trait* is just a method dictionary in which some method names may be bound instead to $\top$, representing a conflict. Traits may be constructed using the operators $+$ (composition), $-$ (exclusion), $\triangleright$ (overriding) and $[\rightarrow]$ (aliasing). The key point is that traits are always composed using the composition operator $+$, which is associative and commutative [6], hence insensitive to the order in which traits are composed. Conflicts are resolved by the composing class by overriding or excluding the conflicts [17]. We shall use this framework for flattening FTJ.

The flattening property simply states that we can always evaluate the trait composition operators occurring within a class definition to obtain an equivalent class whose method dictionary does not refer to traits — that is, the traits can be compiled away. In order to flatten FTJ programs, we must interpret the parts of the FTJ syntax that represent method dictionaries and traits, and we must define the trait composition operators for those syntactic entities. The translation from FTJ to FJ will simply evaluate the composition operators.

Figure 4 presents the trait composition operators interpreted in the context of FTJ. These operators are used to define the flattening function $[\![\cdot]\!]$ which translates an FTJ class to an

$$[\![\texttt{class C} \lhd \texttt{D uses } \overline{\texttt{TA}} \texttt{ \{}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{;K } \overline{\texttt{M}}\texttt{\}}]\!] \stackrel{\text{def}}{=} \texttt{class C} \lhd \texttt{D \{}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{;K } \overline{\texttt{M}} \rhd [\![\overline{\texttt{TA}}]\!]\texttt{\}} \quad (11)$$

$$[\![\overline{\texttt{TA}}]\!] \stackrel{\text{def}}{=} \sum_{\texttt{TA}_i \in \overline{\texttt{TA}}} [\![\texttt{TA}_i]\!] \quad (12)$$

$$[\![\texttt{T}]\!] \stackrel{\text{def}}{=} \mathit{local}(\texttt{T}) \rhd [\![\mathit{subtraits}(\texttt{T})]\!] \quad (13)$$

$$[\![\texttt{TA with m@n}]\!] \stackrel{\text{def}}{=} [\![\texttt{TA}]\!][n \to m] \quad (14)$$

$$[\![\texttt{TA minus m}]\!] \stackrel{\text{def}}{=} [\![\texttt{TA}]\!] - \texttt{m} \quad (15)$$

Figure 5: Flattening FTJ to FJ

FJ class in Figure 5.

We interpret a sequence of methods $\overline{\texttt{M}}$ as representing a method dictionary, and sequence of trait expressions $\overline{\texttt{TA}}$ as representing a trait composition $\sum_i \texttt{TA}_i$

In order to define the composition operators, we first need a couple of auxiliary functions. $\mathit{lookup}(\texttt{m}, \overline{\texttt{M}})$ (1) returns the declaration of method m in $\overline{\texttt{M}}$, if present. $\perp$ represents an undefined method. $\mathit{extract}(X, \overline{\texttt{M}})$ (2) returns the subsequence of $\overline{\texttt{M}}$ containing the definitions of the methods named in X (where $\bigwedge$ builds a sequence from its operands — if X is empty, then *extract* returns $\bullet$, the empty sequence). $\mathit{mNames}(\overline{\texttt{M}})$ (3) returns the set of method names of methods declared in $\overline{\texttt{M}}$. We will also make use of $\mathit{local}(\texttt{T})$ and $\mathit{subtraits}(\texttt{T})$ (4), which return, respectively, the methods and the subtraits of a named trait T.

The exclusion operator (5) simply removes[2] the definition of m from the method dictionary $\overline{\texttt{M}}$. Overriding (6) removes from $\overline{\texttt{M}}_2$ those methods already defined in $\overline{\texttt{M}}_1$, and concatenates what remains to $\overline{\texttt{M}}_1$. Aliasing (7) simply concatenates an existing method definition for m under the new name n. If, however, the "new" name n is already bound in $\overline{\texttt{M}}$, then a conflict is generated instead. (If m is absent, then we can just ignore the alias, so that any references to n will generate errors.) Note that we have chosen here to represent a conflict by the method body {↑⊥;}. The flattening function will therefore yield a valid FJ program if and only if all conflicts are resolved. (An alternative approach could be to generate FJ code that is syntactically valid, but contains a type error, such as a call to a non-existent method.)

Trait composition is slightly more complicated to define. We first define the auxiliary functions *mBodies* and *broken*. $\mathit{mBodies}(\overline{\texttt{M}}_1, \overline{\texttt{M}}_2)$ (8) represents the method declarations in $\overline{\texttt{M}}_1$ that do not conflict with any methods declared in $\overline{\texttt{M}}_2$. $\overline{\texttt{M}}_1 \cap \overline{\texttt{M}}_2$ represents the method declarations that are (syntactically) *identical* in $\overline{\texttt{M}}_1$ and $\overline{\texttt{M}}_2$ (once again abusing set notation to represent intersection of the method dictionaries). These methods also do not pose any conflicts. $\mathit{broken}(\overline{\texttt{M}}_1, \overline{\texttt{M}}_2)$ (9) represents the set of names of methods with non-identical declarations in both $\overline{\texttt{M}}_1$ and $\overline{\texttt{M}}_2$. These represent actual conflicts. Finally, the composition of $\overline{\texttt{M}}_1$ and $\overline{\texttt{M}}_2$ (10) concatenates the non-conflicting and conflicting method declarations.

Now we are ready to define the translation function $[\![\cdot]\!]$ (Figure 5). A flattened class is

[2]Note that we also adopt the convention initiated by Igarashi *et al.* [8] of using set-based notation for operators over sequences: $M = \texttt{C m(}\overline{\texttt{C}}\ \texttt{x) } \ldots \in \overline{\texttt{M}}$ means that the method declaration M occurs in $\overline{\texttt{M}}$, whereas $\overline{\texttt{M}} \backslash M$ stands for the sequence $\overline{\texttt{M}}$ with M removed. $\overline{\texttt{M}}_1, \overline{\texttt{M}}_2$ is the concatenation of the sequences $\overline{\texttt{M}}_1$ and $\overline{\texttt{M}}_2$. This abuse of notation is justified since the order in which the elements occur in $\overline{\texttt{M}}$ is irrelevant.

one in which its locally defined methods override the (flattened) methods of the used traits (11). Flattening a sequence of FTJ traits or a trait expression always yields a (possibly empty) sequence of FJ methods. A sequence of traits (12) translates to the composition of the translation of its parts. The local methods of a named trait (13) override the composition of its subtraits. Aliasing (14) and exclusion (15) are directly interpreted by the aliasing and exclusion operators.

5 Extending the Traits Calculus with Interfaces and Generics

Although FTJ shows how we can add traits to a simple language like FJ, it does not address any of the issues that we outlined in Section 3. Because FJ is not a real language and does not model features such as concurrency and reflection, it is clearly not an adequate basis to investigate how one could deal with issues related to compilation, reflection, or debugging.

In addition, FJ has only a very limited type system, which means that many of the type related issues discussed in Section 3.2 are also not addressed by FTJ, mainly because FJ does not deal with them either.

In this section we sketch how simple extensions to FJ and FTJ, combined with flattening of traits for these extended languages can lead us to reasonable design choices for these issues.

5.1 Traits and Types

As should be evident from the syntax of FTJ alone, traits in FTJ do not define types. And because FJ and FTJ do not model interfaces, this means that only class names may be used to specify the signature of a method. While this simplifies the theoretical foundation of these models, it poses serious practical problems because it makes it hard or impossible to write traits that can be used across multiple classes.

As an example, consider the trait TLinkable shown in Figure 2. Since in FJ and FTJ, only classes are types, it would not be possible to write this trait in FTJ in a way that is general enough so that it can be reasonably used for multiple classes such as Link and Process.

The method includes:, for example, conceptually takes as its argument an object of any class that uses the trait TLinkable (*e.g.*, Link and Process). But unfortunately, FTJ does not allow us to express this since trait names are not valid types. The only thing that we can do is to use as the argument type either Link or Process, but this also means that the trait can only be reasonably used in the chosen class, and it therefore defeats the purpose of putting the includes: method into a *reusable* trait in the first place!

One way to avoid this problem would be to extend FTJ so that traits, like classes, also define types. In the above example, this means that the trait TLinkable will also define a corresponding type with the same name that can then be used to define the type of the argument in the signature of the method includes:. However, in order for this to work, we also need to extend the definition of subtyping in FTJ so that each class that uses the trait TLinkable is a subtype of the type that is implicitly defined by this trait. And since we want to flatten FTJ programs to FJ, this means that we need to add this form of multiple subtyping also to FJ.

Since we need to extend FJ with a form of multiple subtyping anyway, an alternative approach would be to introduce the notion of interfaces into the calculus. This means that as in Java and C#, each interface defines an FJ type, and classes as well as traits can be declared to be subtypes of numerous interface types. Even though traits themselves cannot

CL	::=	class C ◁ C implements $\overline{\text{I}}$ {$\overline{\text{S}}$ $\overline{\text{f}}$;K $\overline{\text{M}}$}	*Classes*
ID	::=	interface I ◁ $\overline{\text{I}}$ {$\overline{\text{SG}}$}	*Interfaces*
S	::=	C | I	*Types*
SG	::=	S m($\overline{\text{S}}$)	*Method signatures*
K	::=	C($\overline{\text{S}}$ $\overline{\text{f}}$) {super($\overline{\text{f}}$);this.$\overline{\text{f}}$=$\overline{\text{f}}$;}	*Constructors*
M	::=	S m($\overline{\text{S}}$ $\overline{\text{x}}$) {↑e;}	*Methods*
ID	::=	interface I ◁ $\overline{\text{I}}$ {$\overline{\text{SG}}$}	*Interfaces*
e	::=	x | e.f | e.m($\overline{\text{e}}$) | new C($\overline{\text{e}}$) | (S)e	*Expressions*

Figure 6: FJI Syntax.

be used as types, this allows us to solve the identified problem because we can declare a corresponding interface for each trait that should be used as a type. In our example, this means that we declare an interface ILinkable containing the same method signatures as the trait TLinkable, and that we then declare all "linkable classes" (in particular all classes that use the trait TLinkable) as subtypes of ILinkable.

While both approaches, introducing interfaces or using traits as types, require adding multiple subtyping to the calculi, there are important conceptual differences between these two approaches. At the first glance, the approach of treating each trait as a type may seem more convenient in practice, but the presence of exclusions and aliases add a certain complexity to the subtype relation.

Furthermore, making each trait be a type blurs the important conceptual distinction between implementation and interfaces, which leads to several problems in the context of a nominal type system.

- It does not address the fact that in the same way as subclassing does not necessarily imply subtyping [4], a trait may be composed from another trait without conceptually being a subtype of it.
- There may be classes that accidentally conform to the type associated with a certain trait such as TLinkable, but they do not actually use this trait because they follow a different implementation strategy.
- If there are multiple traits providing different implementations of the same conceptual interface, we end up with multiple identical types.

To avoid these problems we will use an approach where traits do not define types, and we use interfaces instead. This is this approach that has been followed by Denier and Cointe in their implementation of traits with AspectJ [5].

5.2 FJI and FTJI

We will first extend FJ with interfaces, obtaining Featherweight Java with Interfaces (FJI). Then we define Featherweight-Trait Java with Interfaces (FTJI) as an extension of FTJ.

The calculus FJI is rather trivial to define. Figure 6 shows the syntax of FJI. The semantics of FJI is almost identical to that of FJ. The rules for *Small-step operational semantics*

$$\frac{}{\texttt{S<:S}} \qquad \frac{\texttt{class C} \lhd \texttt{D implements } \overline{\texttt{I}}\ \{\overline{\texttt{S}}\ \overline{\texttt{f}};\texttt{K}\ \overline{\texttt{M}}\}}{\texttt{C<:D} \quad \forall i.\texttt{C<:I}_i}$$

$$\frac{\texttt{S<:S}' \quad \texttt{S}'\texttt{<:S}''}{\texttt{S<:S}''} \qquad \frac{\texttt{interface I} \lhd \overline{\texttt{I}}\ \{\overline{\texttt{SG}}\}}{\forall i.\texttt{I<:I}_i}$$

Figure 7: FJI Subtyping.

CL	::=	$\texttt{class C} \lhd \texttt{C uses } \overline{\texttt{TA}} \texttt{ implements } \overline{\texttt{I}}\ \{\overline{\texttt{S}}\ \overline{\texttt{f}};\texttt{K}\ \overline{\texttt{M}}\}$	*Classes*
TL	::=	$\texttt{trait T uses } \overline{\texttt{TA}} \texttt{ implements } \overline{\texttt{I}}\ \{\overline{\texttt{M}}\}$	*Traits*

TA is as in Figure 3 and ID, S, SG, K, M, ID, and e are as in Figure 6.

Figure 8: FTJI Syntax.

and *Congruence* are unchanged. The rules for *Field lookup*, *Method body lookup*, *Expression typing* and *Class typing* require only trivial changes to reflect the new syntax for classes and types. Finally, the rules for *Subtyping*, *Method type lookup* and *Method typing* require straightforward extensions to accommodate the fact that interface definitions introduce new types. As an example, we show the new subtyping rules for FJI in Figure 7.

We show a possible syntax for FTJI in Figure 8. Most of it is as before in FTJ and FJI, with the difference that classes and traits can now both use traits and implement interfaces.

What does this imply for flattening? The answer is given in Figure 9, which shows the new flattening function. We flatten classes as before, expanding the methods of all used traits. The flattened classes additionally implement all the interfaces that are implemented by any of the used traits. Note that these interfaces are not affected by aliases and exclusions; *i.e.*, the flattened classes always implement the interfaces exactly as they occur in the used traits. This is important because aliases and exclusions are used to glue together the *implementations* provided by multiple traits, and because we decided for a strict separation between implementation and types, this should therefore have no effects on the types.

In FTJI, we can now create an interface ILinkable that contains declarations for all the necessary methods of the linkable type and is then used as the argument type of methods such as includes: in the trait TLinkable. In addition, we have to declare all linkable classes to be a subtype of ILinkable. One way of doing this is to explicitly implement the ILinkable interface in all these classes. Alternatively, one could implement the interface ILinkable directly in the trait TLinkable, which means that all classes using this trait will be a subtype of ILinkable without having to explicitly declare it.

5.3 FGJ and FTGJ

While multiple subtyping allows us to define the signature of the method includes so that it is not specific to a single class, FTJ still suffers from a lack of expressiveness when it comes to defining reusable trait methods. As we have pointed out in Section 3.2, this is because the

$$[\![\texttt{class C} \lhd \texttt{D uses } \overline{\texttt{TA}} \quad \texttt{implements } \overline{\texttt{I}}\ \texttt{\{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;K } \overline{\texttt{M}}\texttt{\}}]\!] \stackrel{\text{def}}{=} \begin{array}{l}\texttt{class C} \lhd \texttt{D} \\ \quad \texttt{implements } \overline{\texttt{I}}\ \mathit{interfaces}(\overline{\texttt{TA}}) \\ \quad \texttt{\{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;K } \overline{\texttt{M}} \rhd [\![\overline{\texttt{TA}}]\!]\texttt{\}}\end{array} \tag{16}$$

$$\mathit{interfaces}(\overline{\texttt{TA}}) \stackrel{\text{def}}{=} \bigwedge_i \mathit{interfaces}(\texttt{TA}_i) \tag{17}$$

$$\mathit{interfaces}(\texttt{T}) \stackrel{\text{def}}{=} \overline{\texttt{I}}\ \mathit{interfaces}(\overline{\texttt{TA}}) \quad \mathit{where}\ \texttt{trait T uses } \overline{\texttt{TA}} \texttt{ implements } \overline{\texttt{I}}\ \texttt{\{}\overline{\texttt{M}}\texttt{\}} \tag{18}$$

$$\begin{array}{rcl}\mathit{interfaces}(\texttt{TA with \{m@n\}}) & \stackrel{\text{def}}{=} & \mathit{interfaces}(\texttt{TA}) \\ \mathit{interfaces}(\texttt{TA minus \{m\}}) & \stackrel{\text{def}}{=} & \mathit{interfaces}(\texttt{TA})\end{array} \tag{19}$$

The translation of $\overline{\texttt{TA}}$ is the same as in Figure 5.

Figure 9: A possible flattening of FTJI to FJI

return types of methods such as reverse and getNext are in fact parametric; *i.e.*, they depend on the class to which the trait TLinkable is finally applied.

Therefore, using an interface such as ILinkable as the return type does not solve our problem because it would only allow the subset of methods specified in ILinkable — rather than the set of all public methods — to be called on the return values. For similar reasons, using an interface for the argument of the methods setNext: and checkAndSetNext: as well as for to the temporary variables used in the method reverse is not an appropriate solution.

This problem can be addressed by extending FTJ with a generics mechanism such as that of Generic Java (GJ) [3], recently introduced in Java 1.5. Using generics, we can write the trait TLinkable with a type parameter that is then used for the return values, the arguments, and the temporary variables of these methods. And whenever the trait TLinkable is applied to a class such as Link and Process, we can then pass the type associated with this class as the concrete parameter (see Section 6 for the corresponding code using C# generics).

In their paper about FJ, Igarashi *et al.* also present the calculus Featherweight Generic Java (FGJ) [7], an extension of FJ that models Java with generics. Following the augmentation from FJ to FGJ, we now define the new calculus FTGJ, which is an extension of FTJ with generics. We then show how FTGJ can be mapped to FGJ by defining an extended version of the flattening function from FTJ to FJ shown in Figure 5.

The syntax of FTGJ is shown in Figure 10. The metavariable `X` ranges over type variables, `S` ranges over types, and `N` ranges over nonvariable types (types other than type variables). As in FGJ, we write $\overline{\texttt{X}}$ as a shorthand for $\texttt{X}_1, \ldots \texttt{X}_n$ (and similarly for `S` and `N`), and assume sequences of type variables contain no duplicate names. We also allow `C<>`, `T<>`, and `m<>` to be abbreviated as `C`, `T`, and `m`, respectively.

The syntactic extension from FTJ to FTGJ is now analogous to the syntactic extension from FJ to FGJ. In particular, class definitions, trait definitions, and method definitions include generic type parameters.

Once the FTGJ syntax is defined, we can now define the flattening-based translation

CL	::=	`class C<`$\overline{\texttt{X}}$` ◁ `$\overline{\texttt{N}}$`> ◁ N {`$\overline{\texttt{S}}$` `$\overline{\texttt{f}}$`;K `$\overline{\texttt{M}}$` `$\overline{\texttt{TA}}$`}`	*Classes*
TL	::=	`trait T<`$\overline{\texttt{X}}$` ◁ `$\overline{\texttt{N}}$`> is {`$\overline{\texttt{M}}$`;`$\overline{\texttt{TA}}$`}`	*Traits*
TA	::=	`T<`$\overline{\texttt{S}}$`> | TA with {m@m} | TA minus {m}`	*Trait expressions*
K	::=	`C(`$\overline{\texttt{S}}$` `$\overline{\texttt{f}}$`) {super(`$\overline{\texttt{f}}$`);this.`$\overline{\texttt{f}}$`=`$\overline{\texttt{f}}$`;}`	*Constructors*
M	::=	`<`$\overline{\texttt{X}}$` ◁ `$\overline{\texttt{N}}$`> S m(`$\overline{\texttt{S}}$` `$\overline{\texttt{x}}$`) {↑e;}`	*Methods*
e	::=	`x | e.f | e.m<`$\overline{\texttt{S}}$`>(`$\overline{\texttt{e}}$`) | new N(`$\overline{\texttt{e}}$`) | (N)e`	*Expressions*
S	::=	`X | N`	*Types*
N	::=	`C<`$\overline{\texttt{S}}$`>`	*Nonvariable types*

Figure 10: FTGJ Syntax.

$$[\![\texttt{class C<}\overline{\texttt{X}}\texttt{ ◁ }\overline{\texttt{N}}\texttt{> ◁ N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;K }\overline{\texttt{M}}\ \overline{\texttt{TA}}\texttt{\}}]\!] \stackrel{\text{def}}{=} \texttt{class C<}\overline{\texttt{X}}\texttt{ ◁ }\overline{\texttt{N}}\texttt{> ◁ N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;K }\overline{\texttt{M}}\texttt{ ▷ }[\![\overline{\texttt{TA}}]\!]\texttt{\}} \quad (20)$$

$$[\![\overline{\texttt{TA}}]\!] \stackrel{\text{def}}{=} \sum_{\texttt{TA}_i \in \overline{\texttt{TA}}} [\![\texttt{TA}_i]\!] \quad (21)$$

$$[\![\texttt{T<}\overline{\texttt{S}}\texttt{>}]\!] \stackrel{\text{def}}{=} \mathit{local}(\texttt{T}, \overline{\texttt{S}}) \triangleright [\![\mathit{subtraits}(\texttt{T}, \overline{\texttt{S}})]\!] \quad (22)$$

$$[\![\texttt{TA with m@n}]\!] \stackrel{\text{def}}{=} [\![\texttt{TA}]\!][n{\rightarrow}m] \quad (23)$$

$$[\![\texttt{TA minus m}]\!] \stackrel{\text{def}}{=} [\![\texttt{TA}]\!] - \texttt{m} \quad (24)$$

Figure 11: Flattening FTGJ to FGJ

$$lookup(\mathtt{m}, \overline{\mathtt{M}}) \stackrel{\text{def}}{=} \begin{cases} M & \text{if } M = \mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>\ S\ m(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\ \{}\uparrow\mathtt{e;\}} \in \overline{\mathtt{M}} \\ \bot & \text{otherwise} \end{cases} \tag{25}$$

$$\frac{\mathtt{trait\ T<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>\ is\ \{}\overline{\mathtt{M}}\mathtt{;}\overline{\mathtt{TA}}\mathtt{\}}}{local(\mathtt{T}, \overline{\mathtt{S}}) = [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\overline{\mathtt{M}} \qquad subtraits(\mathtt{T}, \overline{\mathtt{S}}) = [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\overline{\mathtt{TA}}} \tag{26}$$

$$\overline{\mathtt{M}}[\mathtt{n}\to\mathtt{m}] \stackrel{\text{def}}{=} \begin{cases} (\overline{\mathtt{M}} \setminus lookup(\mathtt{n}, \overline{\mathtt{M}})), conflict(\mathtt{n}) & \text{if } lookup(\mathtt{n}, \overline{\mathtt{M}}) \neq \bot \\ \overline{\mathtt{M}}, \mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>\ S\ n(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\ \{}\uparrow\mathtt{e;\}} & \text{if } \mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>\ S\ m(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\ \{}\uparrow\mathtt{e;\}} \in \overline{\mathtt{M}} \\ \overline{\mathtt{M}} & \text{otherwise} \end{cases} \tag{27}$$

$$\text{where } conflict(\mathtt{m}) = \mathtt{Object\ m()\ \{}\uparrow\bot\mathtt{;\}}$$

Figure 12: Adapted composition operators for FTGJ

from FTGJ to FGJ. This translation is shown in Figure 11. Before we go through the details of the definitions, it is important to note that this translation does not perform any type checks. Consequently, this translation produces an FGJ program for *any* FTGJ program; the generated FGJ program may however be invalid due to inconsistent use of types[3]. Because traits are compiled away in the translation, this means in particular that the bounds of the type parameters of traits are not taken into account. This has the effect that all type parameters in trait definitions are actually unbound; a native type system for FTGJ, however, would use these bounds to perform type-checking of generic traits.

A comparison to the translation from FTJ to FJ (see Figure 5) shows that only the cases (20) and (22) are changed. While (20) reflects the extended class definition syntax of FTGJ, the change in (22) was necessary because a trait `T` that occurs in $\overline{\mathtt{TA}}$ now takes a sequence $\overline{\mathtt{S}}$ of concrete type parameters. This sequence is then passed as a second argument to an extended form of composition operators *local* and *subtraits*.

Figure 12 defines these two operators together with all the other composition operators from Figure 4 that needed to be adapted. The most interesting case is (26), where we extend the rule defining *local* and *subtraits* so that they take two arguments `T` and $\overline{\mathtt{S}}$, and then replace the formal parameters in `T` and its subtraits with $\overline{\mathtt{S}}$ before they return, respectively, the methods and the subtraits of `T`. As in FGJ, replacing the formal type parameters is done using a simultaneous substitution. The other two definitions (25) and (27) are the same as in Figure 4, except that we use the method syntax of FTGJ instead of FTJ.

6 Applying the Flattening Approach to C#

Here, we examine how the theoretical results developed above can be applied to C#. A prototype implementation has been realized as a Bachelors project [14].

The code for the example of Figure 1 with this approach is shown in Figure 13. Note

[3]This means that our translation has a character similar to that of C++ templates, which are only type-checked after being instantiated.

```
trait TLinkable<T> implements ILinkable
{
    public boolean includes(ILinkable other) {
        if (this == other) return true;
        T tail = list.getNext();
        if (tail == null) return false;
        return tail.includes(other);
    }

    public boolean checkAndSetNext(T other) {
        if (other.includes(this)) return false;
        setNext(other);
        return true;
    }

    public T reverse() {
        T result = null;
        T list = this;
        while (list != null) {
            T temp = list.getNext();
            list.setNext(result);
            result = list;
            list = temp;
        }
        return result;
    }

    public abstract T getNext();
    public abstract void setNext(T other);
}

class Link uses TLinkable<Link> {
    ...
}

class Process uses TLinkable<Process> {
    ...
}
```

Figure 13: The generic trait TLinkable used in the classes Link and Process

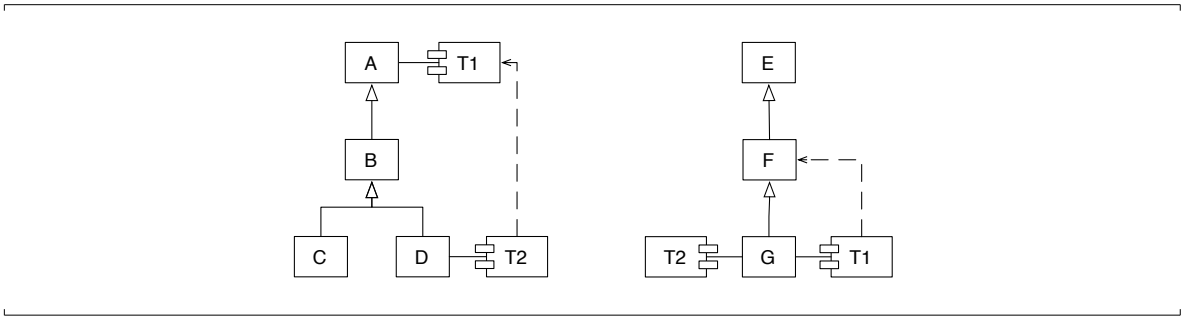

Figure 14: Classes: A-G; Traits: T1, T2. A possible overriding is visualized by dashed lines.

that it uses the Java 1.5 syntax extended with the keywords trait and uses, which are used respectively to declare a trait and to apply a trait.

6.1 Differences between Calculus and Real Language

Although adding traits to FJ is relatively simple, it is another matter to apply traits to a real language like C#. Many language specific aspects have to be taken into account when introducing traits to the language. Some differences and problems are shown in the following paragraphs. More details and some possible solutions have been demonstrated and worked out in a prototype implementation [14].

6.1.1 Modifiers

As is the case for many other statically typed languages, C# defines a couple of modifiers to further control and specify access to state and behavior. However, certain modifiers and maybe even the concept of modifiers don't sit well with traits and cause some problems.

Inheritance (overriding/hiding) Suppose we have a class hierarchy as shown in Figure 14. While this poses no problem in typed languages without explicit inheritance modifiers (*e.g.* Java), it is problematic in C# as it requires the inheritance modifiers virtual, overriding and new. However the explicit behavior expressed by these modifiers cannot be be fully used when using traits. The reason is that methods cannot be declared overriding and override-able at the same time.

As an example, Trait T1 in the right hierarchy in Figure 14 might implement a method explicitly declared overriding. However this trait cannot be used like that in the left hierarchy as T1 doesn't override anything there, therefore requires either no modifier or virtual. Applying overriding would cause compilation errors. We could simply overcome this issue by declaring all T1's methods virtual. However this changes both implementation's behavior or requirements.

Declaring all trait methods to be virtual by default would be a simple solution but it bypasses the explicit modifier concept of C#. As traits are not supposed to interfere with the existing language or change the implementation's behavior this is not a satisfying solution.

Accessibility Similar to inheritance modifiers is the problem raised by accessibility, based on the explicit modifiers public, private, protected, internal and protected internal. Again, assume a class/trait constellation as shown in Figure 14. Not all behavior a trait provides is supposed to be declared by the same modifiers, *e.g.* public. On the contrary other modifiers might also be appropriate. Furthermore, overriding trait methods must reuse the original modifier as accessibility cannot change when overriding methods of super-classes. While these are no issues for using traits it is problematic and inconvenient for the developer as he must not only keep track about the right modifiers within the class hierarchy but also about the ones of all used traits and trait compositions. Therefore maintenance effort and complexity increases for modifiers.

6.1.2 Typing Traits

There are many different kinds of types and type situations we need to take care of when introducing traits to a statically typed object oriented language. The following abstract code example shows some typical typing situations. Notice, the type $\mathbf{T}_x$ must not necessarily be the same type for all type situations.

```
trait TSequenceable {
    public T_return Reverse() {
        T_temporary,return reversedList = new T_instantiation();
        ...
        return reversedList;
    }
    public void Concat(T_argument c1, T_argument c2) {...}
}
```

Using concrete types (Section 5.1) or interfaces (Section 5.2) for traits is very simple. However writing generic and reusable code is difficult, in most cases even impossible. This could lead to further code or trait duplication. The application of such traits is therefore rather limited and not reasonable in general. Furthermore the limited reusability and its consequences might contradict the concept of traits.

Introducing type parameters (Section 5.3) to traits as shown in the example of Figure 13 helps us to code more generic, flexible and reusable traits. However, the parameters can be used to address multiple type situations. Maybe it is even necessary to introduce multiple parameters to address all of them as one parameter might not cover all type situations at once. That way we could use traits like templates. However this introduces some complexity as the developers has to keep track of the way the type parameters are used within the trait's implementation.

As an example, in Figure 13 the type parameter is used everywhere the same way but only to address the type of the class the trait is attached to ("self-reference"). We could also use the type parameter to address any other type or type situation.

A problem happens when classes using parameterized traits have generic type parameters on their own as it might be the case in C#. Using type parameters as a "self-reference" like in Figure 13 is still possible and would cause the following lines to be coded.

```
trait TLinkable<T> { ...}
```

```
class Link<T> uses TLinkable<Link<T>> {... }
```

This is a rather complicated way to reference the class the trait is applied to. The reason is the generic class Link <T> cannot simply be addressed by Link like in Figure 13 as that would reference a completely different class. Besides, in this example it is not possible to use the generic type parameter of the class within the trait — which might be necessary in some cases. Nevertheless, like this we could use the same traits for generic and for non-generic classes in an uniform way.

If we need "access" to the generic type parameters of the class within the trait, we could define an additional parameter for the trait. However this is not a very clean an uniform solution causing confusion about the application of parameters. On the other hand, we could also define a trait to expect a generic class as type parameter.

```
trait TLinkable<S<T>> { ...}
class Link<T> uses TLinkable<Link<T>> {... }
```

However this introduces even more complexity about declaration. Besides the reuse of such traits would be limited again and they couldn't been used by non-generic classes.

Another possible solution is to have parameterized traits having a placeholder to enable a "self-reference". In that case, we suggest to use the identifier of the trait or a keyword like selftype.

```
trait TSequenceable<T> {
    public TSequenceable Reverse() { ... return TSequenceable; }
    public void Concat(TSequenceable c1, TSequenceable c2) {...}
}
```

Like this we are able to use a "self-reference" (but this is not mandatory) and also reuse the generic type parameters of a generic class. This leads to a uniform application of the type parameters for generic applications. However someone might also "abuse" the type parameter to refer to any other type, using the trait like a template. Although this promotes high flexibility the resulting trait model is a bit confusing as types and type parameters might be used in many ways. Furthermore traits might not be shared among generic and non-generic classes either.

6.2 Implementation

The C# language and compilers are rather complex and the application of the calculus to C# causes some troubles (as described in the previous section). Therefore we decided to do the implementation of traits in C# as a pre-processor based on an inline expansion without modifying the language or any compiler. This way the resulting extension in C#, called C#T, is a very simple implementation and stays flexible and open to changes, easily. As a prerequisite we implemented a trait-aware but heavily simplified and stripped-down C# parser.

6.2.1 Traits and trait composition

To enable traits in regular C# classes, we first added a uses-declaration to the class body to reference traits and to provide aliasing (→) and exclusion (^) for conflict resolution as shown in the following example.

```
class Circle : IShape {
    uses {
        TColor { equals(IColor) → colorEquals(IColor); ^equals; };
        TCircle;
    }
}
```

Although this solution doesn't follow typical syntax declaration rules of statically typed languages, it is simple and suits well any typed language. Especially as it doesn't cause any further "overload" of type declarations as this is often the case in such languages.

As an example, assume the generic class Link from Figure 13 having generic parameters, maybe some constraints on these. Besides it might extend a class or implement interfaces, using multiple traits with aliasing or exclusion. The resulting type declaration would be rather hard to "decipher".

```
class Link<T> extends AbstractLink implements iLink
        uses TLinkable<Link<T>> { reverse() → reverseLink; ... } ... { ... }
```

Furthermore traits are defined exactly the same like classes, using a newly introduced unique keyword trait instead of class. A trait might contain a uses-declaration to enable trait composition as well as a requires-declaration to specify requirements towards the class. The body of a trait is a set of regular non-**abstract** methods and operators (we regard an operator as a special method).

```
trait TCircle {
    uses { TShape; }
    requires { double radius(); }
        ...
    }
}
```

Traits can be put into namespaces (like classes) and be placed each into a separate file. A file can even hold multiple traits. However, a file must not contain traits *and* classes (or any other types) at the same time. The syntax for enabling generics is the same as for classes in C# using <...>.

6.2.2 Flattening Traits using inline expansion

As the uses-declaration is designed as a member of the class and not as part of the class' declaration, the inline process only needs to substitute the uses-declaration by the method definitions retrieved from the referenced and flattened traits.

The flattening/inline expansion of traits is realized as a direct recursion on the traits and their uses-declaration, propagating methods and requirements from one level to the next higher one. The process terminates when all traits have been flattened into the class.

A debugger pops-up in case of unsolved trait conflicts, unfulfilled requirements or any other conflict that might be detected during the process. Although the debugger capabilities

are limited it gives some information where and why errors occurred. After a successful preprocessing the resulted source code contains the trait-methods and a comment on each. These sources can be compiled using any existing C# compiler. There is no debugging support on traits for the compilation process.

6.2.3 Adaptations for the C#T prototype

Beside enabling plain traits, we decided to add some language-specific aspects to the prototype and adapt these to traits. Some of them are optional, others are necessary to achieve a reasonable implementation.

First of all, the prototype does not only allow classes to use traits but also structs and other C# types. It supports generic and non-generic types. The same for traits. Besides, libraries can be used by traits (in the same way they're used by regular types) and get automatically propagated to types. Furthermore, traits might implement interfaces (Section 5.2), causing the class implementing these interfaces.

However, the prototype does not yet implement an applicable or satisfying solution to the modifier and typing issues mentioned in Section 6.1. For simplicity, type parameters in parameterized traits (generic traits) always refer to the generic type parameters of a generic type and cannot be used for other types. That means, using traits like templates is not possible in the current implementation.

To make the prototype usable despite some issues, we implemented some simple compiler checks to "guarantee" the preprocessing results in compilable code. More details about the implementation are discussed in [14].

6.3 Evaluation

Implementing traits in a statically typed object oriented language like C# is conceptually simple. However the realization proves to be rather problematic as solutions are either ambiguous, non-uniform or rather complex contradicting the concept and simplicity of traits.

Especially typing traits and introducing generics to traits is not trivial. Type parameters enable flexible traits and code reuse. However they also cause difficulties in finding a uniform but simple model for generic and non-generic use. Mostly, simple solutions are not flexible enough and vice versa. Anyway using type parameters increases implementation complexity which somehow contradicts the concept of traits.

There are also issues about modifiers, a common concept for statically typed languages. Accessibility modifiers must be used in an uniform way (*e.g.* public only) as they would cause a higher complexity in the implementation. However this might not fit the concept of accessibility in statically typed object-oriented languages.

Furthermore, overriding and hiding are explicit in C#. Without introducing higher complexity and effort in implementation or declaration, this would cause the developer to declare all methods to be virtual.

As the theory cannot help in finding a satisfying solution, further research has to be done to address the typing and modifiers issues. The prototype implementation now gives us a good and easy tool in trying out various strategies and solutions as well as evaluating these

```
class SyncA : public A
{
    public:
    virtual int read() {
        acquireLock();
        result = A::read();
        releaseLock();
        return result;
    };
    virtual void write(int n) {
        acquireLock();
        A::write(n);
        releaseLock();
    };

    protected:
    virtual void acquireLock() {
        // acquire lock
    };
    virtual void releaseLock() {
        // release lock
    };
};
```

Figure 15: The class SyncA in C++

against real applicability. However, some compromises might be needed to enable traits in statically typed object-oriented languages.

7 Implementing Traits in C++ using MI and Templates

The language C++ [20] is quite unique regarding its composition mechanisms: it features native support for multiple inheritance and also supports mixins by means of classes with a parameterized superclass (*i.e.*, templates). In this section, we first give a brief overview of these composition mechanisms. Then we analyze whether and how it would be possible to express traits and trait composition in C++ as a combination of these two mechanisms.

7.1 Multiple Inheritance in C++

As suggested by its name, C++ multiple inheritance allows a class to inherit from more than one (direct) superclass. As with traits, the C++ multiple inheritance operation is symmetric, which means that all the (direct) superclasses of a class have the same precedence and conflicts have to be resolved explicitly.

A distinctive feature of multiple inheritance in C++ is that the programmer has a certain amount of control over a diamond situation. If a base class (that is, a superclass) is declared to be *virtual*, the base class is shared and attributes are inherited only once[4].

[4]In his description of C++ [20], Stroustrup uses the term "mixin" for a class that overrides methods of a

```
class SyncReadWrite
{
    public:
    virtual int read() {
        acquireLock();
        result = directRead();
        releaseLock();
        return result;
    };
    virtual void write(int n) {
        acquireLock();
        directWrite(n);
        releaseLock();
    };

    protected:
    virtual void acquireLock() {
        // acquire lock
    };
    virtual void releaseLock() {
        // release lock
    };

    virtual int directRead() = 0;
    virtual void directWrite(int n) = 0;
};
```

Figure 16: The class SyncReadWrite implemented with two abstract methods

While this provides help for avoiding conflicts and ambiguities in a diamond situation, it does not help us to solve the problem of factoring out generic wrappers [17, 15]. This means that with C++ multiple inheritance, it is difficult to factor out wrapper methods (*i.e.*, methods that extend other methods with additional functionality) as reusable classes.

As an example, assume that a class A implements two methods read and write that provide unsynchronized access to some data. If it becomes necessary to synchronize access, we can create a class SyncA that inherits from A and *wraps* the methods read and write. That is, SyncA defines new read and write methods that call the inherited methods under control of a lock. Figure 15 shows the implementation of SyncA in C++.

Now suppose that class A is part of a framework that also contains another class B with read and write methods, and that we want to use the same technique to create a synchronized version of B. Naturally, we would like to factor out the synchronization code so that it can be reused in both SyncA and SyncB.

With multiple inheritance, sharing code among different classes means (directly or indirectly) inheriting from a common superclass that contains the code to be shared. Therefore, if we want to share the synchronization code in SyncA to create another synchronized subclass SyncB of B, we need to factor this code into a new class SyncReadWrite and then make it the

virtual base class. This definition of "mixin" differs from that used in this paper and in most of the research literature.

```
class SyncA : public A, SyncReadWrite
{
    public:
    virtual int read() {
        return SyncReadWrite::read();
    };
    virtual void write(int n) {
        SyncReadWrite::write(n);
    };

    protected:
    virtual int directRead() {
        return A::read();
    };
    virtual void directWrite(n) {
        A::write(n);
    };
};
```

```
class SyncB : public B, SyncReadWrite
{
    public:
    virtual int read()
        return SyncReadWrite::read();
    };
    virtual void write(int n) {
        SyncReadWrite::write(n);
    };

    protected:
    virtual int directRead() {
        return B::read();
    };
    virtual int directWrite(n) {
        B::write(n);
    };
};
```

Figure 17: Code duplication in the classes SyncA and SyncB

superclass of both SyncA and SyncB.

Unfortunately, multiple inheritance alone is not expressive enough to do this. The problem is that the calls to the superclass versions of read and write are statically bound and can refer only to a *superclass* of SyncReadWrite. Therefore, the class SyncReadWrite cannot explicitly call the unsynchronized versions of the methods read and write provided by its *subclasses* A and B.

As a workaround, one would have to modify the methods read and write in SyncReadWrite so that the explicit calls to the superclass methods are replaced by calls to abstract methods directRead and directWrite (Figure 16), which will then be implemented by the subclasses SyncA and SyncB (Figure 17). This solution is still far from satisfactory, since it requires duplication of four glue methods in each subclass. Furthermore, avoiding name clashes between the synchronized and unsynchronized versions of the read and write methods makes this approach rather clumsy, and one has to make sure that the unsynchronized methods directRead are not publicly available in SyncA and SyncB.

7.2 Template-based Mixins in C++

Unlike the generics mechanisms of most other languages such as Java and C#, the C++ template mechanism allows the programmer to write classes with generic superclasses. As shown by VanHilst and Notkin [21, 22] as well as Smaragdakis and Batory [18, 19], this enables the programmer to express a mixin as a class with a generic superclass. Thus, the C++ programmer can avoid the limitation of multiple inheritance with regard to wrappers by using mixins instead.

In the previous example, this means that the synchronization code can be written as a generic class MSyncReadWrite. This generic class can then be used to create the classes SyncA and SyncB by applying it to the superclasses A and B, respectively. The corresponding code

```
template <class Super>
class MSyncReadWrite : public Super {
    public:
    virtual int read() {
        acquireLock();
        result = Super::read();
        releaseLock();
        return result;
    };
    virtual void write(int n) {
        acquireLock();
        Super::Write(n);
        releaseLock();
    };

    protected:
    virtual void acquireLock() {
        // acquire lock
    };
    virtual void releaseLock() {
        // release lock
    };
};

class SyncA : public MSyncReadWrite<A> {};

class SyncB : public MSyncReadWrite<B> {};
```

Figure 18: Synchronization expressed as a mixin

```
template <class Super>
class MLogOpenClose : public Super {
    public:
    virtual void open() {
        Super::open();
        log("Opened");
    };
    virtual void close() {
        Super::close();
        log("Closed");
    };
    virtual void reset() {
        // reset logger
    };

    protected:
    virtual void log(char* s) {
        // write to log
    };
};

class MyDocument : public MSyncReadWrite<MLogOpenClose<Document>> {};
```

Figure 19: The class MyDocument built from two mixins

is shown in Figure 18.

Apart from the fact that C++ mixins are explicitly written as generic classes, this approach is identical to ordinary mixins []. Therefore, it is not surprising that it solves our problem without any code duplication, but also suffers from the linearization problems pointed out previously as soon as multiple mixins are composed.

As an example, assume that we want to combine the mixin MSyncReadWrite with another wrapper mixin MLogOpenClose to create a new class MyDocument, which differs from its superclass Document in that it synchronizes all the calls to the methods read and write, and logs all the calls to the methods open and close. Unfortunately, this requires the programmer to choose an order for the two mixins. In the code shown in Figure 19, we decided to apply the mixin MSyncReadWrite last, which means that it overrides all the features of the other mixin MLogOpenClose. This is not a problem as long as the two mixins do not implement conflicting features. But it does make the whole hierarchy fragile with respect to changes: if the mixin MSyncReadWrite is changed so that it also provides a method reset, then this new method will *implicitly* override the implementation provided by MLogOpenClose and hence break our class MyDocument.

7.3 Traits in C++

Traits resulted from the attempt to design a composition mechanism that combines the beneficial properties of both multiple inheritance and mixins. Since both these mechanisms are supported in C++, this poses the interesting question whether it is possible to express traits and trait compositon in C++ by combining multiple inheritance with templates.

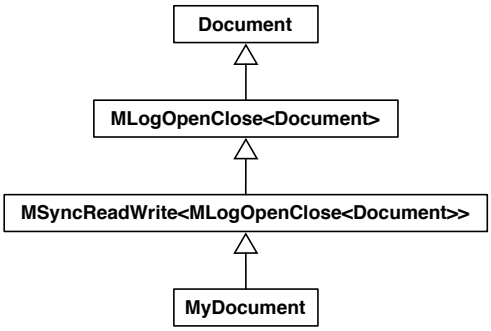

Figure 20: Using C++ templates to simulate mixin composition

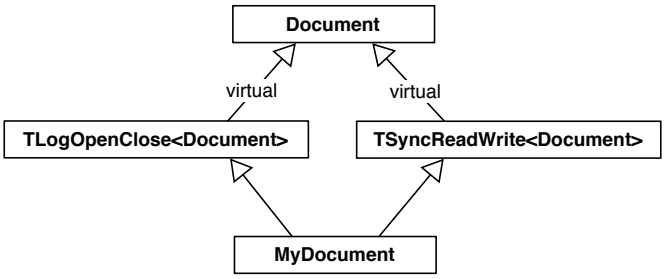

Figure 21: Using C++ templates and virtual base classes to simulate trait composition

It turns out that this is indeed possible. The trick is that instead of expressing the reusable entities as generic classes and composing them into a linear inheritance hierarchy by template instantiation as suggested by VanHilst and Notkin [21, 22] as well as by Smaragdakis and Batory [18, 19], we express them as classes with a *virtual* generic base class and then compose them into a *parallel* hierarchy using multiple inheritance.

The conceptual difference between these two approaches is illustrated in Figures 20 and 21. Figure 20 shows how the class MyDocument is derived from the class Document by a nested instantiation of the templates MLogOpenClose and MSyncReadWrite, which leads to a linear hierarchy. In contrast, Figure 21 shows how MyDocument is built from two templates TLogOpenClose and TSyncReadWrite, which are both applied to the class Document and are then composed using multiple inheritance.

The implementation of the template-based traits TSyncReadWrite and TLogOpenClose is shown in Figure 22. Since the method bodies are omitted in the figure, it is important to note that the methods in TSyncReadWrite are identical to the ones in the mixin MSyncReadWrite (Figure 18). In fact, the only difference between C++ mixins and the corresponding traits is that the traits declare their generic base classes to be virtual.

Declaring the base class to be virtual is crucial as it would otherwise not be possible to correctly compose the traits using multiple inheritance. This is because composing these two traits means instantiating them with the *same* base class Document and then combining them using multiple inheritance. According to the semantics of virtual base classes [20],

```
template <class Super>
class TLogOpenClose : virtual public Super {
    public:
    virtual void open() { ... };
    virtual void close() { ... };
    virtual void reset() { ... };

    protected:
    virtual void log(String s) { ... };
};

template <class Super>
class TSyncReadWrite : virtual public Super {
    public:
    virtual int read() { ... };
    virtual void write(int n) { ... };
    protected:
    virtual void acquireLock() { ... };
    virtual void releaseLock() { ... };
};

class MyDocument : public TLogOpenClose<Document>,
                   public TReadWriteSync<Document> {
    ...   // glue methods
};
```

Figure 22: Implementing MyDocument as the composition of two "C++ traits"

the resulting diamond situation has the key properties known from traits: the common base class Document is inherited only once, and methods in the traits TLogOpenClose and TSyncReadWrite override methods inherited from the common base class Document, while they are overridden by methods in the common subclass MyDocument. Furthermore, methods that are implemented by both traits TLogOpenClose and TSyncReadWrite result in a conflict that needs to be resolved in the subclass MyDocument.

C++ allows one to express composite traits by nesting the templates that represent traits. As an example, we can write a new trait TLogAndSync as a template class that is parameterized by `super` and inherits from the virtual base classes TLogOpenClose and TReadWriteSync, which are both instantiated with the new parameter `super`.

```
template <class Super>
class TLogAndSync : virtual public TLogOpenClose<Super>,
                    virtual public TReadWriteSync<Super> {};
```

A difference between traits and their C++ approximation is the fact that C++ supports only one of the three composition operators of traits: it can express trait sum (+) but not alias (→) or exclusion (−). Whereas aliases can be simulated by disciplined use of the scope modifier ::, this is not the case for exclusion. This means that instead of excluding one or more conflicting methods from a composition, C++ requires the programmer to resolve every conflict by overriding the conflicting methods. While this may result in the same runtime behavior, it is not equivalent from a compositional point of view. When using exclusion, the introduction of a new conflicting method always leads to a conflict that requires explicit resolution, for example by excluding the new method. This is not the case for overriding in C++, where a newly occurring conflict is implicitly overridden by the old conflict resolution code.

7.4 Evaluation

Implementation Characteristics. Generalizing the above findings, we can say that C++ allows one to express trait-like composition by using a combination of nested templates and multiple inheritance with virtual base classes. But what does this kind of trait implementation mean for the issues we outlined in Section 3?

Since traits are represented as template classes with a virtual, parameterized superclass, they essentially have the same properties as all other template classes in C++. Since C++ does not have binary run-time extensibility, traits can't be distributed and linked as a library. The source code of traits must be present at compile time in their entirety.

Consequently, C++ traits cannot be compiled separately; instead, each trait is compiled in context of all the classes where it is actually used. This means that using a trait in many different classes can lead to a significant duplication in object-code, even though many compilers try to reduce this overhead by compiling only the trait members that are actually used in a certain class.

Also regarding error detection at compile time, traits inherit the properties of templates. Since there is no separate compilation of traits, many errors in a trait are only detected when it is actually applied to a class. Compilers tell the programmer whether an error occurred in a trait (template) or in a class that uses the trait. However, as for all templates, traits suffer from the problem that error messages are often quite verbose and hard to understand.

Since C++ templates are not types, neither are traits. This is quite crucial because it means that it is completely up to the programmer to establish appropriate subtype relations. One way of doing this is to represent trait types as abstract classes and to declare subtype relations by (multiply) inheriting from these classes. This is similar as the approach described in Section 5.2, where we suggest a complete separation between the implementation of traits and types. In particular, if a trait is declared to be a subtype of a certain type (by inheriting from the corresponding abstract class), all classes that use this trait are subtypes of this type as well.

Since many of the recent C++ debuggers support debugging of templates, C++ traits can be debugged as well. This means that any of the debugging operations such as setting breakpoints can be used for traits in the same way as they are used for classes.

Conceptual Complexity. Besides the fact that C++ traits inherit some of the problematic characteristics associated with C++ template classes, the approach of simulating traits with C++ templates and multiple (virtual) inheritance also leads to more conceptual problems when compared with traits as a stand-alone composition mechanism.

We first observe that although C++ does not support the complete set of trait composition features, expressing traits in C++ can be achieved only by using a quite sophisticated combination of advanced language mechanisms such as nested templates and virtual base classes. As a consequence, using traits in C++ not only requires one to have a deep understanding of these mechanisms, but it also requires a lot of coding discipline to achieve the robustness benefits promised by the traits mechanism. As an example, the programmer has to avoid using nested scope modifiers (*e.g.*, Super::Super::reset()) to avoid fragility with respect to (distant) changes in the class and trait hierarchies. Similarly, one has to factor out all direct accesses to an overridden trait method into a *single* accessor method that is then called from all the other methods that require access to the overridden functionality. This avoids the fragility that arises if explicit calls to trait methods (*e.g.*, TColor::rgb()) are scattered throughout the source code of multiple methods.

The intrinsic complexity may be part of the reasons why this particular combination of C++ mechanisms was, to the best of our knowledge, not previously identified and suggested as a general composition idiom in C++. This is similar to template-based mixins in C++, which were scientifically investigated and described by VanHilst and Notkin [22, 21] as well as by Smaragdakis and Batory [18, 19] only after mixins were proposed as a fundamental composition mechanism by Moon [11] and later analyzed by Bracha and Cook [2]. As noted by VanHilst [22], templates were previously used, for example in the C++ Standard Template Library (STL) [12], for genericity (*i.e.*, writing data structures such as collections that can be used in the context of different types), but not for role composition using inheritance.

8 Implementing Traits in Squeak/Smalltalk

Our primary implementation of traits is in the Smalltalk dialect Squeak. Because Squeak is dynamically typed, reflective, and purely object-oriented, such an implementation is quite different from what we described so far. Although these differences make it hard to directly adapt the Squeak implementation strategy to less dynamic languages, it certainly does not hurt to understand the basic principles of this implementation. Also, it is interesting to see how the Squeak implementation deals with the issues stated in Section 3.

8.1 Implementation Overview

Squeak is a purely object-oriented programming language based on Smalltalk-80. This means in particular that all Squeak language concepts such as classes and methods are first class objects; they are instance of classes and can be manipulated in the exact same way as all other objects. As a consequence, the task of adding traits to Squeak could be performed entirely in Squeak itself, and it consisted of two parts. First, we extended the kernel so that it can represent traits and trait composition. Then we made sure that instances of classes composed from traits exhibit the runtime behavior specified by the traits model.

Representing traits and trait composition. To make sure that classes can actually be composed from traits, we first extended the implementation of classes to include an additional instance variable to contain the information in the composition clause. This variable defines the traits used by the class, as well as any exclusions and aliases. Based on this extended definition of classes, we then also introduced a representation for traits. This means that a trait is essentially stripped-down class that can define neither state nor a superclass.

Flattening traits at composition time. In Squeak, each class holds a reference to a *method dictionary*, which is a hash table that includes all the class' methods in form of bindings form a method selector to the actual method object containing the byte-code. Together with a class' superclass, the method dictionary is what the virtual machine uses to perform a method lookup. Whenever a message is sent, the lookup algorithm first checks whether the message selector can be found in the message dictionary of the receiver's class. If so, the lookup returns the associated method. Otherwise, the lookup continues in the class' superclass (or terminates if there is no superclass).

Based on this infrastructure, we achieved the correct runtime behavior by flattening the traits structure of each class at composition time. This allows us to ignore the traits structure at runtime and use the ordinary method lookup algorithm that takes only the (flattened) method dictionaries of classes into account.

The flattening process affects the method dictionary of a class C that is composed from at least one trait as follows.

- The method dictionary of C is extended with an entry for each provided trait method that is not excluded, is not overridden in C, and does not conflict with another method.
- For each alias that does not conflict with another method, we add to the method dictionary of C a second entry that associates the new name with the aliased method.
- For each conflicting method, we add to the method dictionary of C an entry that associates the method selector with a special method representing a method conflict.

Since compiled methods in traits do not usually depend on the location where they are used, the CompiledMethod objects (*i.e.*, the byte-code) can be shared between the trait that defines the method and all the classes and traits that use it. The only exception is the methods that use the keyword `super` because they store an explicit reference to the superclass in their literal frame. Therefore, these methods need to be copied with the entry for the superclass changed appropriately. This copying could be avoided by modifying the virtual machine to that it computes `super` when needed, rather than reading it from the literal table for the method.

8.2 Evaluation

Our experience with Squeak shows that implementing traits in a purely object-oriented and dynamically typed single inheritance language like Squeak is unproblematic. Although our implementation is quite straight-forward it avoids most of the issues stated in Section 3.

One reason for this is that we introduced a first-class representation of traits in Squeak. This does not only allow us to compile traits separately, but it also leads to a full integration of traits into the reflective infrastructure of Squeak. This makes it possible to ask, for example, what traits are used by a class (or another trait), what methods are provided and required in a trait, and what trait methods are aliased or excluded in a certain composition. Because the Squeak debugger is based on the reflective infrastructure, making the debugger traits-aware is straight-forward and only requires minor changes (mainly of the debugger's user interface).

Because Squeak methods and method dictionaries are first class objects, and because methods that do not directly access class or instance variables are usually independent of their class, most trait methods can be shared among all the classes (and traits) where they are used. (For the reason explained above, the only exception are methods containing sends to `super`, which are relatively rare.)

This leads to a nearly ideal solution for the trade-off between program size and execution speed: while only a very small percentage of the code in traits needs to be duplicated, a program with traits exhibits the same performance as the corresponding single inheritance program in which all the methods provided by traits are implemented directly in the classes that use those traits[5]. This is especially remarkable because our implementation did not introduce any changes to the Squeak virtual machine, which lead to a very small engineering effort.

Finally, because Squeak is a dynamically typed language, all the issues related to static types are avoided in the first place.

9 Other Implementations

9.1 Traits in Scala

Traits are a built-in language mechanism of the language Scala [13], a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. The traits adaptation of Scala is particularly interesting as Scala is a statically typed language with a type system similar to the ones of Java and C#. In the current version, Scala programs can be compiled either to Java or the .NET platform.

9.1.1 Declaring and Composing Traits.

Scala traits are modeled as abstract classes that do not encapsulate state, neither in the form of variable definitions nor by providing a constructor with parameters Consequently, trait declarations have the same form as class declarations except that the keyword class is replaced by the keyword trait. As an example, consider the definition of a trait Emptiness providing a method isEmpty, which is defined in terms of an abstract method size:

[5]The only performance penalty results from the use of accessor methods, but such methods are in any case widely used in Smalltalk because they improve maintainability. Modern JIT compilers routinely inline accessors, so we feel that requiring their use is now entirely justifiable.

```
trait Emptiness {
   def isEmpty: Boolean = size == 0;
   def size: int;
}
```

In order to apply a number of traits to a class, Scala offers the optional with declaration that follows the class name and an optional extends declaration specifying the class' superclass. As an illustration, consider the definition of a class IntSet that inherits from ScalaObject and uses the two traits Emptiness and Testing:

```
class IntSet extends ScalaObject with Emptiness with Testing {
   ...
}
```

The semantics of **with** T1 **with** T2 is the same as that of uses T1 + T2 in the Squeak implementation (and our formal model []): the composition is symmetric (*i.e.*, the order of the traits does not matter) and conflicts need to be resolved explicitly. Note, however, that conflict resolution is less flexible in Scala because it does not feature exclusion and aliasing.

An interesting feature of Scala is that traits cannot only be composed but can also be inherited, which is a consequence of the fact that Scala traits are just special classes. This means that both classes and traits can be defined as an extension of other traits. For example, Scala allows one to define a trait B that inherits from a trait A and uses the two traits U and V:

```
trait B extends A with U with V {
   ...
}
```

The semantics of this construct is the same as if A and B were classes: local methods in B override methods in U and V, which in turn override methods inherited from A. This form of trait inheritance allows the programmer to establish partially ordered compositions of traits: features of trait B override all equally named features of trait A. It also allows the programmer to use the keyword `super` in the trait B to access (overridden) methods defined in A, which somewhat compensates for the missing alias operator.

In this context, it is important to note that in Scala, a super-send super.foo() that occurs in a trait B is only valid if B inherits from another trait A that implements (or inherits) the method foo. If the method foo inherited from A is abstract, the super-send super.foo() in B has the semantics known from super-sends in our traits: it will refer to the method foo in the superclass of the class to which B will eventually be applied.

9.1.2 Integration into the Type System.

The most interesting aspect of the Scala adaptation of traits is the fact that they are fully integrated into Scala's static type system. Because Scala traits are modeled as a abstract classes, each trait, like each class, also defines a type. This is important because it means that in Scala, traits without any concrete methods play the roles of interfaces, and Scala therefore does not have a separate notion of interfaces.

The Scala type system supports generics in a similar but even more expressive way than Java 1.5. This is important for the integration of traits, because it allows the programmer to express generic traits without having to introduce any trait-specific additions to the type

system. Instead, generic traits are written in exactly the same way as are generic classes. As an example, consider the fully abstract trait Set, which is parameterized with a type parameter T that corresponds to the type of the set's elements:

```
trait Set[T] {
  def includes(x: T): Boolean;
  def add(x: T): Set[T];
  ...
}
```

9.1.3 Implementation Characteristics.

In the current version of Scala, traits cannot be compiled separately; instead, traits are compiled together with the classes where they are used. This means on one hand that the object-code of traits is duplicated in all these classes. It also means that whenever a class uses a trait, the actual source code of the trait must be available at compile time.

Currently, Scala programs are debugged using debuggers of their native platforms (Java and .NET). Because traits (as well as the other Scala specific features) are compiled away at that level, these features are not directly represented when debugging. The same holds for reflection, which is not yet supported on the level of Scala.

As far as the issues regarding static typing are concerned, the integration of traits in Scala is quite pragmatic. The basic idea is that traits are just a special form of abstract classes, which cannot encapsulate state, neither in the form of variable definitions nor by providing a constructor with parameters. Consequently, traits "inherit" most of the characteristics of classes and these two concepts can be handled in a very uniform way.

In particular, this means that each trait defines a type and that trait composition defines a subtype relationship on these types. (Note that Scala avoids the problems related to the question of how the subtype relation should be defined in presence of aliases and exclusions because these operators are not supported in Scala.) Because traits enable a form of multiple subtyping in a very similar way as interfaces do in Java or C#, Scala does not support the notion of interfaces. This means that instead of interfaces, the Scala programmer can use traits that contain only abstract methods.

While this unification of interfaces and traits simplified the language by reducing the number of language concepts, it somewhat blurs the conceptual difference between implementation and interfaces and can lead to the problems outlined in Section 5.1. Together with the fact that Scala traits can be generic, this unification leads to a very practical solution that avoids most of the expressiveness issues stated in Section 3.2.

9.2 Traits in VisualWorks

VW Traits is an implementation of traits for VisualWorks [23] by Terry Raymond. Although VisualWorks is like Squeak a dialect of Smalltalk, there are essential differences between VW Traits and our Squeak based implementation of traits. A major difference is that VW Traits can include state, which means that they can specify instance variables, class instance variables, and shared variables. When a trait is used in a class, the variables defined in the trait will be added to the ones defined in the class. However, unlike with methods, identically named variables are unified and do not cause conflicts. If a class C uses two traits T1 and T2 that both specify a variable x, only one variable x is added to the class, and all references

to x in methods of T1 and T2 are bound to this variable. Similarly, if C defines or inherits a variable x and uses a trait that defines another variable x, these two variables are unified.

While unification of variables makes the use of traits defining variables very straightforward, it is somewhat problematic because it can easily lead to unexpected behavior when a class uses two traits that provide identically named variables that are used for different purposes. Because the variables are unified rather than causing a conflict, this problem may not be detected at composition time.

Another distinctive feature of VW Traits is that they support "policy objects" that determine what action to take when particular composition situations occur. By defining additional policies, the programmer can for example specify how to resolve certain conflicts. A drawback of VW Traits is that they do not support the alias operator, which makes it had to implement certain glue methods without code duplication.

Regarding the implementation, the biggest difference between VW Traits and traits in Squeak is that VW Traits do not reuse CompiledMethod objects. Instead, the source code of a trait method is copied to each client class and recompiled in the class when the trait is installed. This means that both the source code and the byte-code of trait methods are duplicated when the trait is applied to a class. The reason for this implementation strategy is twofold. First, the VisualWorks virtual machine does not permit one to execute a method that has not been compiled specifically for the class of the receiver. Second, because methods in a VW Trait can contain instance variable references, they need to be recompiled to update the instance variable offsets.

VW Traits are merged into client classes using a "trait specification". Unlike our composition clause, which is part of the class definition, this trait specification is a "pragma method" that identifies the trait to be merged and the package that is to contain the merged methods. The trait specification allows the programmer to exclude certain trait methods by adding them to the list of excluded methods.

9.3 Traits in Perl 5

Inspired by our initial publication on traits [17], Stevan Little ported traits to Perl 5. His implementation closely conforms to the description of traits in our first publication []. In particular, it supports all three trait composition operators (sum, alias, exclusion), it allows one to express required methods, and it requires one to explicitly resolve all method conflicts.

Unlike the purely object-based language Smalltalk, Perl 5 is not fundamentally object-oriented. Instead, Perl models objects as references that know what class they belong to. Classes are expressed as packages, and methods are subroutines that expect an object reference to the receiver as the first argument.

Following these principles, Perl traits are also expressed as packages. As an example consider Figure 23, which shows the Perl implementation of the trait TSyncReadWrite (*cf.* the corresponding C++ implementation is discussed in Section 7.1 and shown in Figure 16). On the first line, we begin the trait definition by declaring the package where the trait resides (*i.e.*, the name of the trait). The second line declares this package to be a trait by using the package base from the module Class::Trait. This is necessary in order to be able to properly resolve all method calls. After declaring the requirements, the trait implements the synchronized versions of the methods read and write.

Using this trait the synchronized class SyncA can be derived from the base class A as shown in Figure 24. (See Section 7.1 for the corresponding implementations using C++ templates.)

```
package TSyncReadWrite;

use Class::Trait 'base';

our @REQUIRES = qw(read write);

sub read {
    my ($self) = @_;
    $self->acquireLock();
    my $result = $self->SUPER::read();
    $self->releaseLock();
    return $result;
}

sub write {
    my ($self, $n) = @_;
    $self->acquireLock();
    $self->SUPER::write($n);
    $self->releaseLock();
}

sub acquireLock() { ... };

sub releaseLock() { ... };
```

Figure 23: The trait TSyncRead in Perl

Again, the first line starts the class definition by declaring a new package that is named after the class. Then, we declare that the new class SyncA inherits from the base class A using the trait TSyncReadWrite.

The fundamental differences between Perl and Smalltalk is reflected in the actual implementation of traits in these two languages. A detailed description of the Perl implementation is outside the scope of this paper. We note only that like the Smalltalk implementation, the Perl implementation is also based on flattening the trait structure at compile-time: *i.e.*, for each relevant trait method, the class' symbol table is extended with an entry that refers to the original trait method.

9.4 Traits in Fortress

The Fortress Programming Language is a general-purpose, statically checked, nominally typed, component-based programming language designed for producing robust high-performance software with high programmer productivity.

9.4.1 The Fortress Object Model.

The Fortress object model has two basic concepts: object and trait. As in most other object-oriented programming languages, a Fortress object consists of fields and methods. The fields of an object are specified in its definition. An object definition may also include additional

```
package SyncA;

use base ("A");

use Class::Trait ("TSyncReadWrite");
```

Figure 24: The class SyncA defined as a subclass of A using the trait TSyncReadWrite

method definitions.

A Fortress trait is a named program construct that declares a set of methods. A method may be either abstract or concrete: abstract methods have only headers; concrete methods also have bodies. A trait may extend an arbitrary number of other traits: it inherits the methods declared by the traits it extends (except those that it overrides).

Every object has a set of traits; it inherits the concrete methods of its traits (except those that are overridden) and must include a definition for any abstract method declared by any of its traits.

As in Scala, each Fortress trait also represents a type. However, because there are no classes in Fortress, traits are the only entities that define types. That said, it is important to mention that also every defined Fortress object has an associated type. This is because every object implicitly defines a trait (of the same name), of which it is an instance. The trait implicitly defined by an object includes, as abstract methods, all of the public methods, including all implicitly defined public accessors, introduced by the object definition (i.e., those methods not declared by any traits of the object). It also extends all of the declared traits of the object.

Another similarity between Fortress and Scala is that both languages do not support operators for aliasing or excluding trait methods form a composition. However, Fortress allows a trait to exclude another *trait*. If a trait T excludes another trait S, the two traits are mutually exclusive: no object can have them both, no third trait can extend them both, and neither may extend the other. Similarly, a Fortress trait T may include a *bounds* clause, which has the meaning that the trait must not be extended with immediate subtraits other than those that appear in its bounds clause.

Like Scala, also Fortress does not have a separate language construct for interfaces; it just uses completely abstract traits instead. Being a statically typed language, Fortress also allows method overloading. This means that methods can be overloaded within a trait but the set of methods declared or defined in a trait must be mutually compatible. In particular, it is a static error if a trait contains two methods with identical or ambiguous signatures. As in the original traits proposal, such conflicts must be explicitly resolved.

9.4.2 Discussion.

Because we do not yet have detailed information about the characteristics of the implementation of traits in Fortress, we can only discuss about how Fortress traits address the conceptual issues stated in Section 3.

From a conceptual point of view, the Fortress adaptation of traits is quite unique as it is

not an extension of a class-based single inheritance model. This means that instead of using traits to realize more fine-grained decompositions of classes in a single inheritance setting, Fortress traits are used as the primary (and only) way of composing objects. Together with the fact that traits are the only types in Fortress, this leads to a very simple and concise object model.

However, the absence of single inheritance also means that only symmetric composition is possible. However, according to our experience, it is sometime useful to have partially ordered compositions, where a wrapper trait (such as SyncReadWrite discussed in Section 7.1) applied to a subclass automatically overrides some dedicated methods inherited inherited from a superclass.

Also, it is not clear whether and how Fortress allows a programmer to refer to an overridden method. Since each object (and each trait) may use multiple traits, a single keyword such as super would not be enough to avoid potential ambiguities, and at the same time, Fortress traits do not seem to support aliases.

As far as the issues regarding static typing are concerned, the integration of traits in Fortress is quite elegant and comparable to the Scala solution. Each trait is also a type and trait composition implies subtyping. Furthermore, Fortress supports parametric polymorphism in a similar way as Scala, which means that one can write traits that are parameterized with various types.

10 Concluding Remarks

This report presents an overview of the problems and issues at stake when integrating traits into a statically typed programming language. A key idea that we promote is that any integration should focus on the flattening property as an acid test for any typed approach to traits. Flattening is not necessary a good implementation strategy, but it is useful during prototyping as it is straightforward to implement.

Although *semantically* traits can be flattened, a proper integration of traits in a given language cannot be achieved by mere syntactic transformation. In our Squeak implementation of traits [9, 15] traits are first-class entities from which classes can be composed. First-class traits enable code reuse. In addition we reuse methods at the level of method dictionaries, by physically sharing common methods among traits and classes, without introducing run-time penalties [17]. Similarly, an extension of a statically typed language with traits should be consistent with flattening, but a robust implementation would require a deeper integration of traits with the host language.

Acknowledgments

We gratefully acknowledge the financial support of Microsoft Research for the project "Traits in C#". We warmly thank Luigi Liquori for his helpful comments and insights. We also thank Arnaud Spiwack, Marcus Denker and Tudor Gîrba for reviewing drafts.

References

[1] A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proceedings OOPSLA'03 (International Conference on Object-Oriented*

Programming Systems, Languages and Applications), volume 38, pages 47–64, Oct. 2003.

[2] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, Oct. 1990.

[3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, pages 183–200. ACM Press, 1998.

[4] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proceedings POPL '90*, San Francisco, Jan. 1990.

[5] S. Denier. Traits programming with AspectJ. In P. Cointe, editor, *Actes de la Première Journée Francophone sur le Développement du Logiciel par Aspects (JFDLPA'04)*, pages 62–78, Paris, France, Sept. 2004. Available at http://www.emn.fr/x-info/obasco/events/jfdlpa04/.

[6] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *Transactions on Programming Languages and Systems*, Mar. 2006. To appear.

[7] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. In *Proceedings OOPSLA '99, ACM SIGPLAN Notices*, pages 132–146, Nov. 1999.

[8] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, May 2001.

[9] A. Lienhard. Bootstrapping Traits. Master's thesis, University of Bern, Nov. 2004.

[10] L. Liquori and A. Spiwack. Adding multiple inheritance to Featherweight Java. INRIA Sophia Antipolis & ENS Cachan, available at www-sop.inria.fr/mirho/Luigi.Liquori/PAPERS/ftj.pdf, 2004.

[11] D. A. Moon. Object-oriented programming with Flavors. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 1–8, Nov. 1986.

[12] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison Wesley, 1996.

[13] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical Report 64, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, 2004.

[14] S. Reichhart. A prototype of traits for C#. Informatikprojekt, University of Bern, 2005.

[15] N. Schärli. *Traits — Composing Classes from Behavioral Building Blocks*. PhD thesis, University of Berne, Feb. 2005.

[16] N. Schärli, S. Ducasse, and O. Nierstrasz. Classes = traits + states + glue (beyond mixins and multiple inheritance). In *Proceedings of the International Workshop on Inheritance*, 2002.

[17] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[18] Y. Smaragdakis and D. Batory. Implementing layered design with mixin layers. In E. Jul, editor, *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 550–570, Brussels, Belgium, July 1998.

[19] Y. Smaragdakis and D. Batory. Mixin-based programming in C++. In *2nd Symposium on Generative and Component-Based Software Engineering (GCSE 2000)*, Erfurt, Germany, 2000.

[20] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.

[21] M. VanHilst and D. Notkin. Using C++ Templates to Implement Role-Based Designs. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37. Springer Verlag, 1996.

[22] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings OOPSLA '96*, pages 359–369. ACM Press, 1996.

[23] Cincom Smalltalk, Sept. 2003. http://www.cincom.com/scripts/smalltalk.dll/.